Electronic Communications of the EASST
Volume 40 (2011)

Proceedings of the 4th International Workshop on Petri Nets and Graph Transformation (PNGT 2010)

Towards Guided Trajectory Exploration of Graph Transformation Systems

Ábel Hegedüs, Ákos Horváth and Dániel Varró

20 pages

Guest Editors: Claudia Ermel, Kathrin Hoffmann
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Towards Guided Trajectory Exploration of Graph Transformation Systems*

Ábel Hegedüs[1], Ákos Horváth[2] and Dániel Varró[3]

[1] hegedusa@mit.bme.hu,
[2] ahorvath@mit.bme.hu
[3] varro@mit.bme.hu
http://inf.mit.bme.hu/en
Department of Measurement and Information Systems (MIT)
Budapest University of Technology and Economics (BME), Budapest, Hungary

Abstract: Graph transformation systems (GTS) are often used for modeling the behavior of complex systems. A common GTS analysis scenario is the exploration of its state space from an initial state to a state adhering to given goals through a proper trajectory. Guided trajectory exploration uses information from some more abstract analysis of the system as hints to reduce the traversed state space. These hints are used to order possible further transitions from a given state (selection) and detect violations early (cut-off), thus pruning unpromising trajectories from the state space. In the current paper, we define cut-off and selection criteria for guiding the trajectory exploration, and use Petri Net analysis results and the dependency relations between rules as hints in our criteria calculation algorithm. The criteria definitions include navigation along dependency relations, various types of ordering for selection and quantifiers for cut-off criteria. Our approach is exemplified on a cloud infrastructure configuration problem.

Keywords: graph transformation; trajectory exploration; Petri nets

1 Introduction

Model transformation is a common technique in Model Driven Engineering to design, analyze and simulate various kinds of models. In case of model analysis, forward transformations usually carry out an abstraction (and create an abstraction gap) to enable efficient formal verification and validation. Increasing the level of abstraction usually increases the efficiency of formal analysis but decreases its preciseness. As a result, mapping the information gathered from validation back to the original models (i.e. *back-annotation*) is also a challenge due to the abstraction gap between the source and target languages. Therefore analysis results in the target model may only serve as a *hint* for the source model, and further processing steps are needed to complete the analysis. For example a hint can provide only the number of operations instead of their original sequence and further steps can obtain the sequence itself.

When analyzing graph transformation systems (GTS), a highly relevant technique in many application areas for modeling the behavior of systems, Petri nets (PN) are often used as an

* This work was partially supported by the SecureChange (ICT-FET-231101) project and the Janos Bolyai Scholarship.

abstraction to perform verification [KK06, BS06, Ren04], optimization [VV06] or find errors in the implementation (debugging) [WKS+09]. However, the results in certain kinds of analysis methods are not always *execution paths* (ordered sequences of rule applications or transition firings) for the GTS, but more abstract information such as an *occurrence vector* containing only the number of transition executions (instead of their exact order). This occurrence vector may serve as a hint for calculating an execution path in the GTS.

In order to successfully retrieve the rule application sequences (execution paths or trajectories) on the GTS-level, we have to explore the states reachable from an initial state by applying available rules. This approach is called *state space exploration* and is often used in verification of graph transformation systems [KK06].

Guided trajectory exploration differs from state space exploration in making use of a hint (obtained, for instance, from some more abstract analysis result) that can reduce the number of states, which are explored to find an adequate trajectory by guiding exploration in the state space.

The challenges of a guided trajectory exploration are two-fold: (1) at every state during the exploration, the hint can be used to decide if the current state is part of an infeasible path, thus we can terminate the exploration along this path (i.e. it is a dead-end in the search of a final state). This decision is made by evaluating a *cut-off criteria*. (2) The hint can be used to select which alternative exploration direction should be explored first (e.g. prioritize the alternate transitions to a next state by their likelihood of leading to the final state). This ordering is made by evaluating a *selection criteria*.

In our paper, we define cut-off and selection criteria for guiding the trajectory exploration, and use the PN analysis results and the dependency relations between rules as hints in our criteria calculation algorithm. The cut-off criteria are defined to exploit the dependencies between graph transformation rules in order to make the decision based on information about the effects of future rule applications thus allowing early termination of infeasible paths. Similarly, selection criteria use the dependencies to be able to calculate how alternative rules can affect the applicability of future rules and prioritize them accordingly. The criteria definitions include navigation along dependency relations, numerical operations on an occurrence vector, various types of ordering for selection and quantifiers for cut-off criteria.

The rest of the paper is structured as follows. First, we give a high-level overview of the complete trajectory calculation approach in Section 2. Section 3 introduces the cloud configuration example, graph transformation systems and their abstraction as Petri nets. We introduce the dependency graph, and explain how it correlates to the state of the GTS during trajectory calculation in Section 4. Section 5 defines selection and cut-off criteria and specifies their calculation algorithm, which is illustrated by the case study. Finally, related work is discussed in Section 6 and Section 7 concludes our paper.

2 Overview of the approach

The guided trajectory exploration problem appears in many scenarios, such as configuring infrastructures or other autonomic software systems. In our paper we deal with scenarios where the states are represented with graphs and operations are defined as GT rules. The rules and the initial graph are represented together as a *GTS* in Figure 1, which illustrates the overview of our

trajectory exploration approach. Apart from the GTS, the *state space exploration* contains a *goal*, which describes properties of the final state (such as the number of elements of a given type), and *constraints* that each state must satisfy along the trajectory we seek (e.g. minimum number of elements of a given type). In our approach, both goals and constraints are defined as graph patterns. Finally, the exploration *strategy* (including cut-off and selection criteria) is used to decide between multiple possible operations at a given state. The overall objective of the approach is to find a *trajectory* from the initial state to the final state. In our approach the strategy uses hints retrieved from analysis of the abstracted GTS as described in the following:

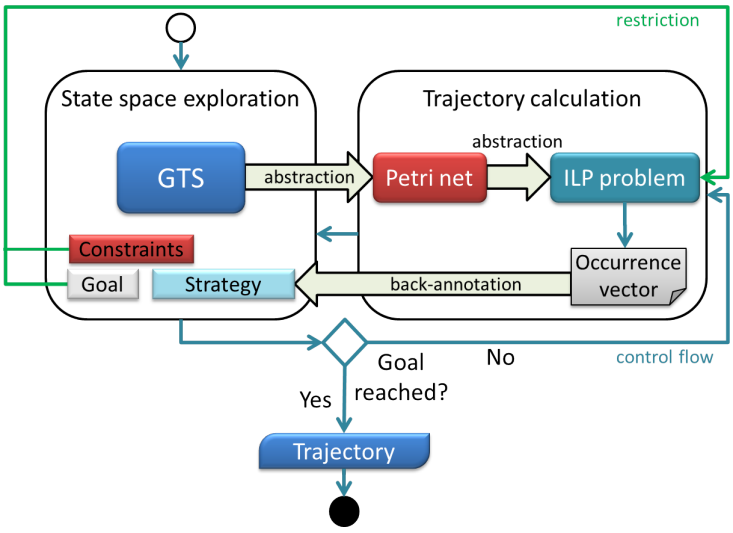

Figure 1: Overview of the solution

Occurrence vector-based search strategy approach In [VV06], the computation of an optimal rule application sequence is performed by encoding the *Petri net* abstraction (detailed in [VVE+06]) of the GTS into an *integer linear programming (ILP) problem*. The *solution* of this problem (solved using CPLEX[1] in our implementation) is a candidate transition occurrence vector. Since the abstraction does not guarantee that this vector corresponds to an executable execution, its feasibility should be checked on the GTS-level. However, in the original approach, the occurrence vector was used in the GTS state space exploration by only allowing occurrence vector compliant execution paths to be explored. Therefore, it did not help in selecting the most promising execution path or cutting the search on infeasible paths.

Selection and cut-off criteria for search strategy In this paper, we propose additional techniques, which also use the occurrence vector as a hint, to guide the state space exploration (implemented as an extension to the VIATRA2 model transformation framework [V2]) to further increase the performance of the algorithm. The main features of these new techniques are (a) using the rule (or transition) *dependency graph* (G_d) computed from the GTS (using the Condor [Con]

[1] http://www.ibm.com/software/integration/optimization/cplex-optimizer/

dependency analyzer tool) to have a global view on the effects of rule applications [MKR06]; (b) defining *selection criteria* Cr^{sel} on the applicable rules (transitions) at a given state (for deciding between alternative upcoming operations); and (c) defining *cut-off criteria* Cr^{cut} on the paths (for deciding about the feasibility of further exploration). Criteria defined in both (b) and (c) depend on G_d and the application numbers for the rules (transitions) in the occurrence vector.

3 Definitions

In this section the basics of graph transformation, GTSs and their PN-based abstraction are shortly discussed. Before the definition, we introduce our demonstrating case study.

3.1 Example

We consider services built on top of a *cloud middleware* (CM) using components as building blocks. *Servers* (S) and high-availability *clusters* (Cl) can be deployed on the CM, while *databases* (DB) are installed on servers and *applications* (App) are executed over databases. Finally, servers can also be deployed on clusters and *storage* (St) subsystems can only operate over clustered servers. Note that the configuration is not a tree structure (e.g. a database is deployed on multiple services, which in turn are deployed on a cloud or cluster), but a directed graph.

In order to provide an appropriate infrastructure for clients, the configuration of the cloud infrastructure must meet certain requirements, e.g. an application and a storage subsystem is required for a cloud-based web service. Such an infrastructure is shown in Figure 2.

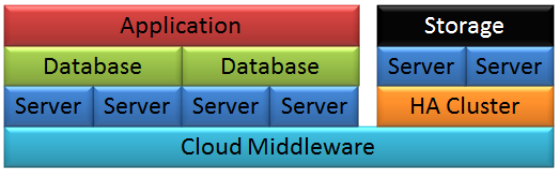

Figure 2: An example system providing reliable service

To satisfy this constraint the cloud configuration has to be designed in an appropriate way. We assume that regular change management commands are issued by some middleware service broker. If the current infrastructure of the cloud implies that the required parameters cannot be satisfied by the actual cloud configuration, reconfiguration operations are to be initiated, which lead the system into a state where all constraints are met.

The reconfiguration actions of cloud components will be captured by a graph transformation system that is defined subsequently. An overview on using graph transformations for software architecture reconfigurations can be found in [BHTV06].

3.2 Graph Transformation

A graph $G = (N, E, src, trg)$ is a 4-tuple with a set N of nodes, a set E of edges, a source and a target function $src, trg : E \to N$. A type graph TG is an ordinary graph. An instance graph G is

typed over TG by a typing morphism $type : G \to TG$. Let $card(G,x)$ denote the cardinality (i.e. the number of graph objects) of type $x \in TG$ in graph G.

An example type graph is shown in Figure 3. The type graph contains only one cloud component *Node* designated graphically as a rectangle. The edges *CM, Cl, S, DB, App* and *St* are used to denote the type of the component such that the source and the target node of this edge is the same node (*So* means *Socket* and represents a *CM* or *Cl*). Edge *onR* connect two different components denoting that the source node is deployed on the target node of this edge.

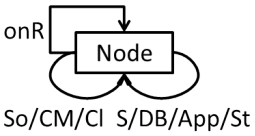

Figure 3: Type graph

Graph transformation (GT) [CMR+97] provides a rule-based manipulation of graph models. A graph transformation (GT) rule typed over a type graph TG is given by $r = (L \xleftarrow{l} K \xrightarrow{r} R)$ where L (left-hand side), K (context) and R (right-hand side) graphs are typed over TG and graph morphisms l, r are injective. The negative application conditions (NAC) of a GT rule are given by a (potentially empty) set of pairs (N,n) with N being a graph also typed over TG and $n : L \to N$ an injective graph morphism.

Application of a rule r to a *host graph* G alters the model graph by replacing the pattern defined by L with the pattern of R. This is performed by (i) *finding a match* of pattern L in model G (ii) *checking the negative application conditions* N, which may prohibit rule application, i.e. if there is a match of N in G (as an extension of the match of L in G), then the rule is not applicable (iii) *removing* a part of the model M that can be mapped to pattern L but not pattern R yielding an intermediate graph D and (iiii) *adding* new elements to the intermediate graph D, which exist in R but not in L yielding the derived graph H.

A *graph transformation sequence (GT sequence)* is a sequence of GT steps (application of a rule on a given match), i.e., a sequence of rule applications. A GT sequence starting from graph G yields G' and more than one GT step may belong to it. In the paper, we follow the *Double Pushout Approach* [CMR+97].

Example 1 *The ongoing example is captured by a set of graph transformation rules in Figure 4. In order to simplify the graphical presentation, we simply write the type* CM, S, Cl, DB, App, St *of the component on the node (which is denoted by a rectangle) instead of self-loop edges. This way, only* onR *edges remain, which we also omit by representing the hierarchy of deployment using vertical stacking of components.*

The addCM *rule adds a new* CM *to the configuration,* addS *creates a new* S *deploying it on top of a* CM *or* Cl*, however, a* Cl *cannot have more than two* S *deployed on it. Rule* addCl *produces a new* Cl *deploying it on top of a* CM*,* addDb *adds a new* DB *deploying it on top of two* S *that have no other* Node *deployed on them,* addApp *creates a new* App *deploying it on top of two* DB *that have no other* Node *deployed on them. Finally,* addSt *adds a new* St *deploying it on top of two* S *that are deployed on the same* Cl *and have no other* Node *deployed on them.*

Graph transition system A *graph transformation system* $GTS = (R, TG)$ consists of a type graph TG and a finite set of graph transformation rules typed over TG. A *graph transition system*

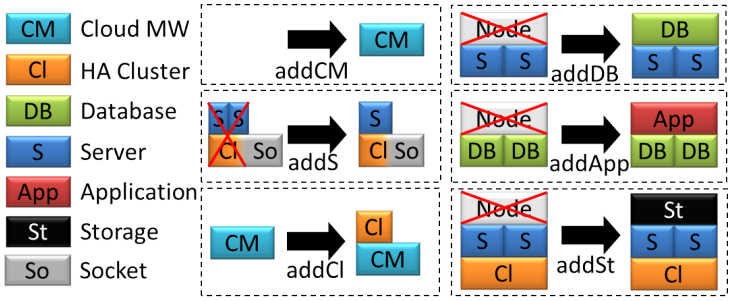

Figure 4: Graph transformation rules

GS is defined as a graph where nodes are instance graphs, and edges are graph transformation steps such that the source and target nodes of the edge are graphs.Starting from G_0 (initial state) the *state space* (i.e. the reachable instance graphs) of *GS* is represented taking into account all applicable rules from a given host graph. The different matches of applicable rules may lead to different edges in *GS*. A path in the graph transition system is a *GT* sequence also called a trajectory between two graphs. Then we say that a graph is reachable from G_0 iff there is a path in the *GS*.

Example 2 *In Figure 5 an extract of the graph transition system of our running example is shown. On the left the root of the graph transition system is the start graph G_0 where the system configuration contains a* CM*, three* S*, and one* DB *components. Rules* addS*,* addCl*, and* addCM *are applicable to G_0, here we follow only the application of the first two rules.*

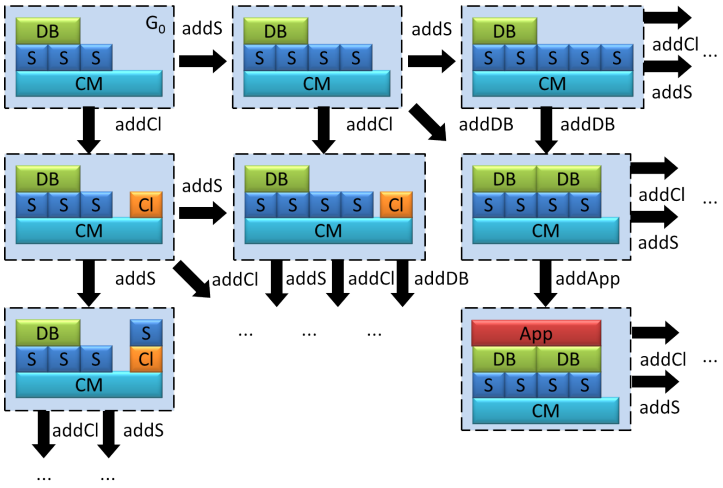

Figure 5: A part of the graph transition system

3.3 Petri Net Abstraction for GTS

Our guided trajectory exploration approach is based on a Petri net abstraction technique introduced for GTS in [VVE+06]. The motivation behind such an abstraction was that solving the reachability problem on the PN level is of much lower complexity than solving the problem directly on the GTS-level using algorithmic exploration techniques.

The essence of this abstraction technique is to derive a cardinality PN, which simulates the original GTS by abstracting from the structure of instance graphs and only counting the number of elements (nodes or edges) of a certain type by placing tokens to a corresponding place. These tokens are circulated by transitions derived from each GT rule, which simulate the effect of the rule on the number of typed elements by adding and removing tokens from corresponding places.

Example 3 *In Figure 6 rule* addCl *of our example in Section 3 is shown with the corresponding type graph on the left. The PN abstraction is shown on the right. According to the type graph of the example, the corresponding cardinality PN has a place for all node types, namely for type* Node, *and edge types, namely* CM, S, Cl, DB, App, St, *and* onR.

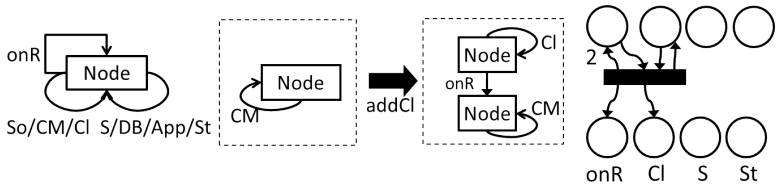

Figure 6: Rule *addCl* and the corresponding cardinality Petri net

For instance, the left–hand side L of rule addCl *contains a node and the edge* CM. *Thus the corresponding transition with the same name has two incoming arcs starting from the corresponding places. Similarly, the right–hand side of the rule consists of two nodes and edges* CM, Cl, *and* onR *thus there are four outgoing arcs to* Node, CM, Cl, *and* onR *with weights* $2,1,1,1$, *respectively.*

In this way whenever rule addCl *is applied the number of the tokens at the involved places changes according to the cardinality of the graph types.*

The incidence matrix of the PN abstraction of the example GTS is in Figure 7. The places (columns) refer to the type places corresponding to the type graph of Figure 6, while transitions (rows) refer to corresponding rules of Figure 4.

W =

	Node	onR	CM	S	Cl	DB	App	St
addCM	1	0	1	0	0	0	0	0
addS	1	1	0	1	0	0	0	0
addCl	1	1	0	0	1	0	0	0
addDB	1	2	0	0	0	1	0	0
addApp	1	2	0	0	0	0	1	0
addSt	1	2	0	0	0	0	0	1

Figure 7: Incidence matrix of the Petri net abstraction

The coverability problem over PN can be encoded into an ILP problem, and the solution of the resulting ILP problem is a *transition occurrence vector* (σ). The transition occurrence vector prescribes how many times a GT rule needs to be applied in order to reach the derived submarking of a solution state. For example, to get from an initial graph containing only one *CM* to a state with four *S* and two *DB*, the shortest solution vector would be $\sigma = \{0,4,0,2,0,0\}$. Further details about the encoding and solving can be found in [VV06].

Note that [VVE+06] proves that the mapping is a proper abstraction in the sense that the derived PN simulates the original GTS, and it also discusses a possible abstraction of NACs into cardinality Petri nets. However, that abstraction would deliver an integer *non-linear* programming problem for the trajectory finding problem for which solution techniques have greater complexity than solution techniques for an (I)LP problem. Thus we ignore the abstraction of NACs in the current paper, but it is important to note that this is not a conceptual restriction since NACs only result in the generation of additional infeasible paths.

4 Definition and Usage of the Dependency Graph

In this section we first describe the notion of graph transformation rule dependency and define the dependency graph that is constructed for GTS (Subsection 4.1). Next, we demonstrate how the dependency graph relates to the state of the GTS during trajectory exploration (Subsection 4.2).

4.1 Graph Transformation Rule Dependency

The application of a GT rule r can alter the graph in a way that other rules, which were disabled before, become enabled (or were enabled and become disabled), thus the application of these rules *depend* on the application of r. The dependencies between rules are independent of the graph they are applied on, and can be derived from their definition. The analysis can be carried out using various techniques, such as graph matching and graph equivalence (critical pair analysis [HKT02]) or unification and backtracking (conditional transformation-based dependency analysis [Con]), and results in a matrix of dependencies between rules.

The result of the analysis is used to create a *dependency graph* (G_d, illustrated in Figure 8) of the rules, where each r_i is a node (n_i) and there is a directed arc from n_i to n_j if r_j has *sequential dependency* on r_i (i.e. the application of r_i may affect the match set of r_j). Note that dependencies introduced by NACs are taken into account as well. Finally, there may be arcs in both directions between two nodes.

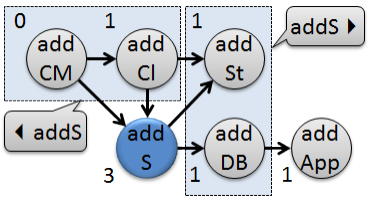

Figure 8: Dependency graph example

In this paper, $n_i \blacktriangleright$ denotes the set of nodes, which have sequential dependency on n_i, while $\blacktriangleleft n_i$ denotes nodes on which n_i has sequential dependency (both sets illustrated for n_{addS} in Figure 8).

Finally, we have a candidate transition occurrence vector (σ) as a solution of the analysis of the PN, where σ(i) is the number of times that r_i is applied during the execution. During the

trajectory calculation, the number of times r_i has been applied in a given path is stored in the *application vector* ($\overline{v}_a$) as $\overline{v}_a$(i). An execution path of the state space exploration is *compliant* with σ if $\overline{v}_a \leq \sigma$ (the number of applications is less or equal for each rule). Throughout the paper we use the difference between σ(i) and $\overline{v}_a$(i) ($\sigma(i) - \overline{v}_a(i)$) as the *remaining application number* of r_i ($\#_i$). This number is stored as an attribute for n_i in G_d together with the *state of* r_i that is either enabled or disabled in a given GTS state.

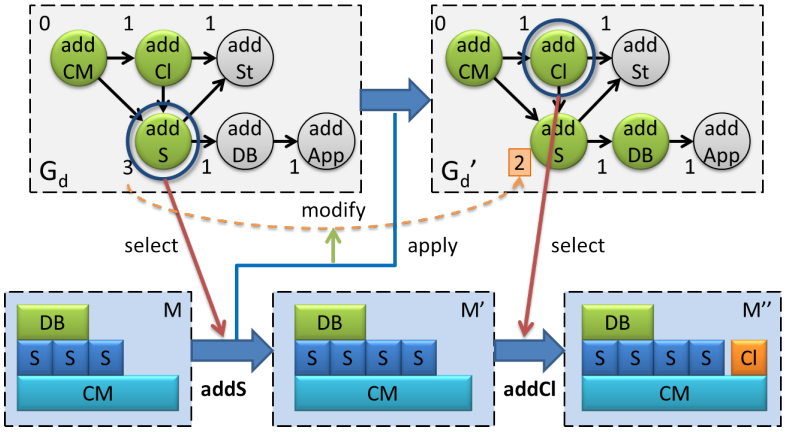

Figure 9: GT rule application and its effects on the dependency graph

4.2 Using the Dependency Graph for Trajectory Exploration

The state of the GTS and the dependency graph are tightly connected for a given initial graph and occurrence vector. Figure 9 illustrates how the application of a GT rule affects the current graph and the dependency graph. First, the current state is depicted as the graph M (representing the current cloud configuration) and dependency graph G_d. The color of the nodes (e.g. n_{addS}) of G_d represent the state of the corresponding GT rules (r_{addS}), green background for enabled, grey for disabled. The number near each node is the remaining application number ($\#_{addS} = 3$).

In the course of trajectory exploration, the next GT rule, which is applied (r_{addS} in the example) is *selected* from the set of enabled rules. The *application* has the following effects on the graphs: (a) graph M changes according to the rule definition (here, a new S is added to CM), the new graph is illustrated as M' (b) the $\#_{addS}$ is *modified* to represent that the rule is applied (it decreases from 3 to 2) (c) G_d is also changed to G'_d, as $\#_{addS}$ decreased and the applicability of GT rules may change (here r_{addDB} becomes enabled). The trajectory exploration then continues from M' by selecting a rule based on G'_d.

5 Definition and Calculation of Cut-off and Selection Criteria

The main novelty in our approach is taking advantage of the dependency relations between graph transformation rules using selection and cut-off criteria, which enhance the trajectory exploration

strategy. In this section we first give an overview of the criteria types (Subsection 5.1), followed by the definition of the criteria constructing building blocks (Subsection 5.2). Finally, we specify an algorithm for calculating arbitrary criteria over dependency graphs (Subsection 5.3) and illustrate its use on the cloud configuration case study (Subsection 5.4).

5.1 Overview of Cut-off and Selection Criteria

Our guided exploration approach uses the dependency graph as additional information to decide in which order the states of the GTS are explored. In a certain state, two decisions are made regarding the next step, (a) whether the current branch is promising and should be further explored, if yes (b) which enabled GT rule should be executed to reach the next state. We define formal criteria over the current dependency graph, which are evaluated in order to support decisions:

- *Cut-off criteria (Cr^{cut})* inspect the current state of the dependency graph and return a boolean result, which is true if further exploration of the current branch cannot lead to a goal state with a compliant trajectory. In this case, the exploration continues from another state instead of executing a GT rule in the current state.
- *Selection criteria (Cr^{sel})* take the dependency graph and define an ordering of the enabled GT rules. A given GT rule r_i is placed before another rule r_j, if the execution of r_i is more promising, based on the criteria and the current state, than the execution of r_j. The ordering is used instead of selecting the most promising rule since the exploration may lead to a cut-off on the most promising branch. In this case the next rule in the ordering is executed.
- *Soft cut-off criteria (Cr^{soft})* differ from regular cut-off criteria by marking a given branch only unpromising, instead of incompatible. In this case the exploration continues from an other branch, but may return to the unpromising branch if a trajectory to a goal state is not found on other branches.

5.2 Criteria Building Blocks

In our approach, the criteria are constructed using starting point identifiers and a well-defined set of operators (i.e. *building blocks*), which can represent navigation over the graph edges, numerical and logical functions between subcriteria, ordering of results and quantifiers. Starting points behave as *operands* and create a criteria together with an operator. The resulting criteria (called a *subcriteria*) can be also an operand to create more complex criteria. Throughout the paper we refer to an operator as *enclosing* for the operands which it is combined with. For the definition of the criteria grammar, see Appendix A. The criteria building blocks are defined in the following:

Starting points identify rules on which a given criteria is interpreted. Trivially, selection criteria are evaluated for rules, which are enabled in the given state and their $\#_i$ is greater than zero. However, cutoff criteria may be defined on rules with different properties (e.g. disabled rules or all enabled rules, regardless of $\#_i$).

Apart from simple starting points such as *enabled* rules ($\mathbb{E}$), *disabled* rules ($\mathbb{D}$), we define numerical *constants* ($\mathbb{C}$) as starting points as well to separate them from operators. Constants are

used in logical and numerical functions when one of the operands is a predefined number. Finally, *custom starting points* ($[Cr]$) are defined when only rules with specific properties are evaluated.

Navigation operators describe which other nodes of graph G_d should be evaluated when starting from a given node. Navigation is defined over the edges between the graph nodes, and can be limited to paths of one or multiple connected edges. During evaluation, the $\#_j$ for each r_j reached by the navigation is summed up, except if navigation occurs in the criterion of a custom starting point, where $\#_j$ is not incorporated in the total.

For a given rule r_i, *forward* navigation ($r_i \blacktriangleright$) returns the set of nodes to which r_i has outgoing edges, while *backward* navigation ($\blacktriangleleft r_i$) returns the set of nodes from which r_i has incoming edges. Furthermore, a given navigation operator can be used iteratively on a set of rules either for a given amount of time (*limited* iteration), e.g. navigating forward twice ($r_i \blacktriangleright \blacktriangleright$) can be defined as $r_i \blacktriangleright^2$. Finally, a given operator can be used iteratively for as long as it is applicable (*transitive* iteration, $r_i \blacktriangleright^+$).

Numerical functions are used when the partial evaluation results of subcriteria are combined. Among the numerical operators, the *addition* operator ($Cr_1 + Cr_2$, where Cr_i are subcriteria) is used most often for summing partial results, but *subtraction* ($Cr_1 - Cr_2$), *multiplication* ($Cr_1 * Cr_2$) and *division* (Cr_1 / Cr_2) are also usable.

Logical functions are also defined between two subcriteria and result in boolean values, where the result depends on the actual operator type and the subcriteria operands. For subcriteria, which have numerical results, the available operators are *equals* ($Cr_1 = Cr_2$), *differs* ($Cr_1 \neq Cr_2$), *more* ($Cr_1 > Cr_2$) and *less* ($Cr_1 < Cr_2$).

Similarly, for subcriteria, which have boolean results (e.g. ones that use one of the operators above), the available operators are *conjunction* ($Cr_1 \wedge Cr_2$), *inclusive disjunction* ($Cr_1 \vee Cr_2$), *exclusive disjunction* ($Cr_1 \oplus Cr_2$) and *negation* ($\neg Cr$).

Ordering and quantifiers are top-level binary operators for selection and cut-off criteria, respectively. Ordering operators define how the numerical results from the subcriteria are returned, with either the highest result being the first (*maximal* operator, $\max Cr$) or the lowest (*minimal* operator, $\min Cr$). For cut-off criteria, we define quantifiers to describe when the subcriteria must be true for at least one rule (*existential* quantifier, $\exists r\ Cr$) or for every rule (*universal* quantifier, $\forall r\ Cr$) among the rules defined by the starting point.

Example 4 *Here, we show the usage of the building blocks using several examples, both for cut-off and selection criteria, which are meaningful when dealing with guided state space exploration. In the rest of the paper, we will illustrate the evaluation of criteria using these examples and the dependency graph from our case study.*

Non-compliant path (Look-ahead) cut-off criterion *When the application of any GT rule would make the current execution path non-compliant with the occurrence vector of its corresponding PN, it can be cut. This criterion does not depend on the dependency graph, and*

can be seen as the only one applied when the guidance is based on the occurrence vector. Equation 1 defines the criterion using the notation introduced in this section.

$$Cr^{cut}_{Ncp} : \forall r \; \mathbb{E}(r) = \mathbb{C}(0) \tag{1}$$

Permanently disabled rule cut-off criterion *The current path can be cut if there is a disabled rule, which still has to be applied based on the transition occurrence vector, but the application of any rule, which it depends on would lead to a non-compliant path. Equation 2 gives the criterion with custom starting points and equally with regular navigation operators.*

$$Cr^{cut}_{Pdr} : \exists r \; \blacktriangleleft [\mathbb{D}(r) > \mathbb{C}(0)] = \mathbb{C}(0) \text{ or equally } \exists r \; (\mathbb{D}(r) > \mathbb{C}(0)) \wedge (\blacktriangleleft \mathbb{D}(r) = \mathbb{C}(0)) \tag{2}$$

Maximal forward-dependant application path selection criterion *Among the applicable GT rules at any given state of the exploration, the one with the most (transitively) dependant rule applications should be executed first (Equation 3). The selection is based on calculating the effect of each applicable rule using the dependency graph and on the idea that a rule, which affects more applications should be applied earlier in the trajectory.*

$$Cr^{sel}_{Mfd} : \max \; \mathbb{E}(r) \blacktriangleright^{+} \tag{3}$$

Minimal backward-dependant application path selection criterion *In order to guide the exploration towards a state where one of the cut-off criteria may be applicable, the rule selection is based on calculating the remaining rule applications for backward-dependant rules (Equation 4). In this case, the calculation starts from rules, which depend on the evaluated rule, therefore the result can informally described as the sum of the remaining application number for rules, which affect the same rules as the current rule.*

$$Cr^{sel}_{mbd} : \min \; \blacktriangleleft^{+} [\mathbb{E}(r) \blacktriangleright] \tag{4}$$

5.3 Calculation Algorithm for Criteria

In Subsection 5.1, we defined selection and cut-off criteria as means to help the decision whether to explore reachable states on a given branch and which GT rules to execute if we do. These criteria are constructed from the building blocks introduced in Subsection 5.2. In this section we specify the algorithm for calculating arbitrary criteria.

Algorithm overview First, we describe the steps of the calculation for arbitrary criteria (Cr) defined using the building blocks. Let us assume that at the beginning of the algorithm we have a dependency graph (G_d) where the status of the nodes (n_i) is updated based on the applicability of the corresponding rule (r_i) and the remaining execution number ($\#_i$) is set based on σ(i) and $\bar{v}_a$(i).

The overview of the algorithm (described as a function over a criterion and a dependency graph) is given in pseudocode here, while a more detailed version of the criteria evaluation algorithm can be found in Appendix B:

function EVALUATE(Cr,G_d)
  initialize variables L_n, S_{st}, S
  check inconsistency of starting points in S_{st}
  if starting points consistent **then**
    initialize S_n with nodes satisfying S
    let $O \leftarrow enclosing(S)$ ▷ list of nodes or cut-off
    for all $n \in S_n$ **do** ▷ Iterate through eligible nodes
      initialize N_c, N_v, R_p, R_b ▷ Current and visited nodes, partial results
      while $O \neq Cr$ **do** ▷ Evaluation terminates when the applied operator is the criteria
        APPLYOPERATOR(O,G_d,N_c,N_v,R_p,R_b,L_n) ▷ Apply operator
        let $O \leftarrow enclosing(O)$ ▷ Get enclosing operator
      end while
      update list or cut-off result
    end for
    return result
  end if
end function

1. Check the set of starting points (S_{st}) in Cr to ensure that there is no apparent inconsistency (e.g. both $\mathbb{E}(r)$ and $\mathbb{D}(r)$ is used).

2. Select starting point S from S_{st}, this selection may use the first simple or custom starting point from the criteria or choose randomly. The selection method does not affect the calculation algorithm, however the selected starting point can greatly affect the required calculation steps for specific criteria (e.g. if a custom starting point would rule out the large majority of rules, the rest of the criteria is not evaluated).

3. Acquire the set of nodes (S_n) from G_d which satisfy S (e.g. nodes for enabled rules for $\mathbb{E}(r)$). If S is a custom starting point, it can be calculated with the same algorithm (starting from **Step 1**, where $S := Cr$) to find satisfying nodes.

4. Select the next node (n) from S_n and the enclosing operator (O) of S. Initialize the set of current nodes (N_c) including only n.

5. Apply O on N_c, where application is based on the type of the operator as follows:

   Navigation operators take the current nodes and return nodes, which are reachable in the graph (in the direction defined by the operator) and are not included in the already visited nodes (N_v). These nodes (N_r) are added to the N_v and will serve as N_c in the next step ($N_c := N_r$). The $\#_i$ of each n_i is summed and added to the partial result (R_p).

   Numerical and logical operators are applied as implied by their definition and result in R_p and boolean values (R_b), respectively.

   Ordering operators place n_s in the appropriate position in the list of calculated nodes (L_n). In case of *maximal* operator, n_s is placed before the first node, which has a lower R_p, and before first node, which has higher in case of *minimal*.

Quantifier operators decide whether Cr^{cut} is satisfied based on R_b. If R_b is true and the quantifier is *existential*, the current branch is cut and other nodes are not calculated. Similarly, if R_b is false and the quantifier is *universal*, the branch is not cut. In both cases, skip to **Step 8**.

6. If there is an enclosing operator O_e for O (if O is transitive and $N_c \neq \emptyset$, $O_e := O$), apply O_e (continue from **Step 5**) on N_c, R_p and R_b (whichever exists).

7. If there is a next node (n_n) in S_n, continue from **Step 4** (i.e. calculate criteria on next node).

8. Return L_n for selection criteria and R_b for cut-off criteria (the branch is cut if true).

Calculation of multiple criteria For more efficient trajectory calculation, several cut-off and selection criteria are defined as the search strategy. The combination of cut-off criteria can be seen as inclusive disjunction (if any of them are true, the branch is cut), while the combination of selection criteria is non-trivial and requires a method for merging different lists.

5.4 Step-by-step Criteria Calculation Example

We illustrate the execution of the algorithm using the *Permanently Disabled Rule* cut-off and *Maximal forward-dependant application path* selection criteria using the cloud configuration case study. Throughout the description of the example we refer back to the steps of the algorithm in parentheses (e.g. **S4** means **Step 4**).

Permanently Disabled Rule The calculation of Cr^{cut}_{Pdr} is illustrated in Figure 10 using the dependency graph of the example with selected $\#_i$ for the rules and the current cloud configuration. Disabled rules are depicted with light gray background (here *addSt* and *addApp*), while enabled rules are drawn with green background. The formal definition of the criteria is also included below the graph with partial result R_p in the bottom right corner. The definition has two starting points, which are consistent (both $\mathbb{D}$, **S1**), thus the algorithm chooses the first one as S (depicted with bold circle, **S2**) then acquires $S_n := \{addSt, addApp\}$ (**S3**) and selects *addSt* as n and $>$ as O (**S4**, 1. in Figure 10).

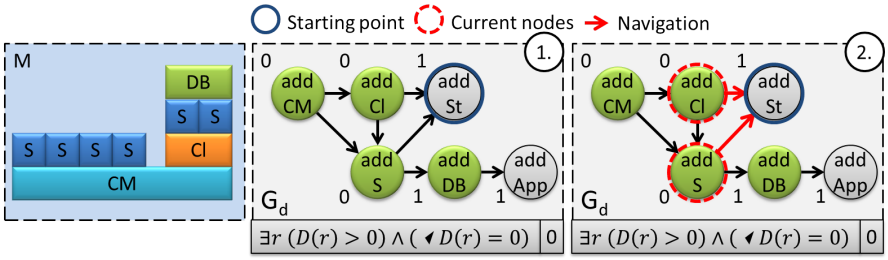

Figure 10: Example evaluation of cut-off

The application of O returns true (**S5**) therefore the enclosing operator $\wedge$ is applied next (**S6**), which means that the second operand has to be evaluated as well. The evaluation of the backward navigation operator ($\blacktriangleleft$) is illustrated in 2. of Figure 10, where the current nodes reached by navigation are depicted with dashed circles (*addCl* and *addSt*, **S5**). Since both $\#_{addCl}$ and $\#_{addS}$ is zero, the $=$ operator evaluates to true (**S6** and **S5**), the original $\wedge$ operator returns true as well (**S5**). Finally, the evaluation of the existential quantifier operator (**S5**) means that the rest of the nodes are skipped and the branch is cut (**S8**).

Maximal forward-dependant application path The calculation of Cr^{sel}_{Mfd} is illustrated in Figure 11 similarly to the first example. The configuration of the cloud and the dependency graph are depicted in a different state, and the formal definition of the criteria is below the graph. As before, the first steps of the calculation algorithm select a starting point ($\mathbb{E}$) from the criteria, the first evaluated node (*addCl*) and the enclosing operator $\blacktriangleright$ (**S1-4**, 1. in Figure 11).

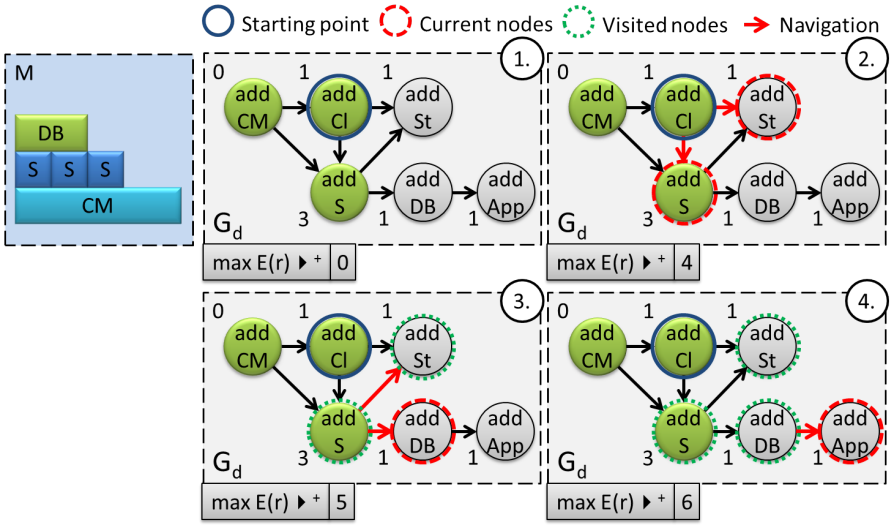

Figure 11: Example evaluation of selection

The nodes reached by the application of the $\blacktriangleright$ operator are $N_c := \{addSt; addS\}$ (2. in Figure 11), and the partial result R_p is updated to 4 (**S5**). Since the enclosing operator is the transitive navigation (**S6**), the $\blacktriangleright$ operator is used in consecutive steps on nodes in N_c. First, *addDB* is reached and R_p is updated to 5 (**S5**, 3. in Figure 11), since *addSt* is already in N_v and no new nodes are reachable from it. Next, *addApp* is reached from *addDB* and R_p is updated to 6 (**S5**, 4. in Figure 11). In the following iteration, N_c is empty (no new reachable nodes), therefore the enclosing operation max is selected (**S6**). The application of this ordering operator puts *addCl* in L_n (**S5**), then *addS* is selected as the next node (**S7**). Once all the nodes are evaluated the ordered list ($L_n = \{addCl, addS\}$) is returned as the final result of the algorithm (**S8**).

It is important to note that this result circumvents a problem when the exploration would apply *addS* as long as possible without applying *addCl* first, which trajectory would not lead to a goal

state if at least a *St* is required in the configuration (or in any configuration where servers have to be deployed on clusters).

Additional Notes on Criteria Calculation The applicability of these criteria highly depends on the structural properties of the dependency graph. First, if most dependencies between rules are bidirectional or if the graph is almost strongly connected, the selection criteria will be less effective. Furthermore, we suggest using transitive closure for path computation as a first approximation, but we believe that more sophisticated algorithms may be defined by handling cycles in the graph differently from simple paths.

6 Related work

The guiding of trajectory exploration based on rule dependency in graph transformation systems is quite novel idea in the field, but similar approaches are not unprecedented in a broader research scope, as described below.

Our trajectory exploration approach can be regarded as an extension of the constraint satisfaction problems over models (abbreviated as CSP(M)) [HV10], which takes GT rules, constraints and goals and searches for solutions that are reachable from an initial model. The criteria calculation defined in our paper serves as a special solver for CSP(M) by substituting regular solvers using simple breadth first search.

Note that our approach is explicitly designed for trajectory exploration and criteria using transformation rule dependency are introduced to increase the efficiency of the state space exploration. The approach in [EGLT11] is similar to our approach as it also exploits the dependencies between GT rules with critical pairs analysis. Here, graph transformation systems are enhanced with control flow as well and the dependency information helps in discovering problems, which could occur in runtime.

Model checking approaches to analyze graph transformation systems are similar to our approach as they also perform state space exploration. One can categorize them as *interpreted approaches* like [BK02, Ren04, KK06], which store system states as graphs and directly apply transformation rules to explore the state space, and *compiled approaches* such as [SDR04, SV03, EJL06, BRRS08, BS06], which translate graphs and graph transformation rules into off-the-shelf model checkers to carry out verification.

GROOVE [Ren04] is a model checker over graph transformation systems. Its main benefit is the ability to verify model transformation and dynamic semantics through applying CTL model checking on the generated state space of the GTS. It is mainly used for modeling and verifying the design-time, compile-time, and run-time structure of object-oriented systems.

Augur2 [KK06] is a GTS model checker that tackles the complexity associated with independent rules by condensing the entire state space into a single graph with unfolding semantics. It also provides some approximative techniques to deal with infinitely large state spaces, and counterexample-guided refinement of this abstraction.

In [KE10] Petri net abstraction is used for verifying graph transformation systems and investigating the reachability problem of forbidden graphs using context-free graph grammars. An important application of the method is deadlock analysis.

The complete approach presented in our paper can also be regarded as a directed model checking approach as categorized by [EJL06]. They use (an ad hoc) SPIN encoding and heuristic search for the analysis of graph transition systems.

Baresi et al. [BRRS08] present a model checking solution for graph transformation systems by translating GTS into Bogor models, which can be used for checking Linear Temporal Logic expressions or special-purpose GT rules similar to GROOVE and CheckVML. In [BS06] graph transformation systems are analyzed using Alloy and its tools, which support property checking and finding trajectories to given final states.

It is common in these solutions that they store system states as graphs and directly apply transformation rules to explore the state space similar to our approach. Their main difference is that they use an exhaustive state space exploration to verify certain conditions in the graph transformation system, while our approach relies on guided traversals.

7 Conclusion and Future Work

Guided trajectory exploration, which is a relevant problem when analyzing graph transformation systems, uses hints to reduce the amount of states traversed when looking for trajectories. The hint is used (i) to decide whether a state is a dead-end (cut-off) and (ii) to order alternative directions to increase the efficiency of the traversal (selection).

In the current paper, we defined selection and cut-off criteria for guided trajectory exploration of GTS, and introduced the dependency graph, which combines GT rule dependencies and occurrence vectors. We also described an algorithm for calculating criteria consisting of navigation and computation over the dependency graph. Our approach was exemplified using the cloud configuration problem.

Future work. We plan to specify and develop the complete guided trajectory exploration technique for arbitrary GTS and evaluate its quality against exhaustive exploration techniques. We aim to support guided trajectory exploration in the VIATRA2 model transformation framework. We are also working on defining more sophisticated (problem-specific) criteria and specialized algorithms to increase the efficiency of the approach.

Bibliography

[BHTV06] L. Baresi, R. Heckel, S. Thöne, D. Varró. Style-Based Modeling and Refinement of Service-Oriented Architectures. *Journal of Software and Systems Modelling* 5, 2006.

[BK02] P. Baldan, B. König. Approximating the Behaviour of Graph Transformation Systems. In Corradini et al. (eds.), *Proc. ICGT 2002: First International Conference on Graph Transformation*. LNCS 2505, pp. 14–29. Springer, Barcelona, Spain, 2002.

[BRRS08] L. Baresi, V. Rafe, A. T. Rahmani, P. Spoletini. An Efficient Solution for Model Checking Graph Transformation Systems. *ENTCS* 213, 2008.

[BS06] L. Baresi, P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Corradini et al. (eds.), *Graph Transformations*. Lecture Notes in Computer Science 4178, pp. 306–320. Springer Berlin / Heidelberg, 2006.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. *In [Roz97]*. Chapter Algebraic Approaches to Graph Transformation — Part I: Basic Concepts and Double Pushout Approach, pp. 163–245. World Scientific, 1997.

[Con] Condor, CT-based dependency analyzer. http://roots.iai.uni-bonn.de/research/condor/.

[EGLT11] C. Ermel, J. Gall, L. Lambers, G. Taentzer. Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency between Control Flow and Functional Behavior. In *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science. Springer-Verlag, 2011. Accepted.

[EJL06] S. Edelkamp, S. Jabbar, A. Lluch-Lafuente. Heuristic Search for the Analysis of Graph Transition Systems. In *Proc. Third International Conference on Graph Transformation*. LNCS 4178, pp. 414–429. Springer, Natal, Brazil, 2006.

[HKT02] R. Heckel, J. M. Küster, G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *In: Proc. ICGT 2002. LNCS*. Springer, 2002.

[HV10] Á. Horváth, D. Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 2010.

[KE10] B. König, J. Esparza. Verification of Graph Transformation Systems with Context-Free Specifications. In Ehrig et al. (eds.), *Graph Transformations*. Lecture Notes in Computer Science 6372, pp. 107–122. Springer Berlin / Heidelberg, 2010.

[KK06] B. König, V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *TACAS*. Pp. 197–211. 2006.

[MKR06] T. Mens, G. Kniesel, O. Runge. Transformation dependency analysis - a comparison of two approaches. In Rousseau et al. (eds.), *LMO*. Hermès Lavoisier, 2006.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Nagl et al. (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. LNCS 3063. Springer-Verlag, 2004.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.

[SDR04] O. M. dos Santos, F. L. Dotti, L. Ribeiro. Verifying Object-Based Graph Grammars. *Electr. Notes Theor. Comput. Sci.* 109:125–136, 2004.

[SV03] Á. Schmidt, D. Varró. CheckVML: A Tool for Model Checking Visual Modeling Languages. In Stevens et al. (eds.), *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*. LNCS 2863, pp. 92–95. Springer, San Francisco, CA, USA, October 20-24 2003.

[V2] VIATRA2 Model Transformation Framework, An Eclipse GMT Subproject. http://www.eclipse.org/gmt/VIATRA2/.

[VV06] S. Varró-Gyapay, D. Varró. Optimization in Graph Transformation Systems Using Petri Net Based Techniques. *Electronic Communications of the EASST (ECEASST)* 2, 2006. Selected papers of Workshop on Petri Nets and Graph Transformations.

[VVE+06] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In *Proc. Third International Conference on Graph Transformation (ICGT 2006)*. LNCS 4178. Springer, Brazil, 2006.

[WKS+09] M. Wimmer, G. Kappel, J. Schoenboeck, A. Kusel, W. Retschitzegger, W. Schwinger. A Petri Net Based Debugging Environment for QVT Relations. In *Automated Software Engineering, 2009. ASE '09.* 2009.

A Criteria grammar definition

The following is the BNF grammar used for defining arbitrary criteria:

⟨*criterion*⟩→ ⟨*cut-off*⟩
⟨*selection*⟩

⟨*cut-off*⟩→ ⟨*quantifier*⟩ ⟨*logical*⟩

⟨*selection*⟩→ ⟨*ordering*⟩ ⟨*navigation*⟩

⟨*quantifier*⟩→ **'∃'** ⟨*rule*⟩
'∀' ⟨*rule*⟩

⟨*ordering*⟩→ **'min'** | **'max'**

⟨*rule*⟩→ **'r'**[a-z]*

⟨*startingPoint*⟩→ ⟨*enabled*⟩
⟨*disabled*⟩
⟨*constant*⟩
⟨*custom*⟩

⟨*logical*⟩→ [⟨*numerical*⟩ | ⟨*startingPoint*⟩] ⟨*logNumOp*⟩ [⟨*numerical*⟩ | ⟨*startingPoint*⟩]
⟨*logical*⟩ ⟨*logBinOp*⟩ ⟨*logical*⟩ | ⟨*logUnOp*⟩ ⟨*logical*⟩

⟨*navigation*⟩→ ⟨*navOp*⟩ [⟨*startingPoint*⟩ | ⟨*navigation*⟩]

⟨*numerical*⟩→ [⟨*navigation*⟩ | ⟨*numercal*⟩] ⟨*numOp*⟩ [⟨*navigation*⟩ | ⟨*numercal*⟩]

⟨*enabled*⟩→ **'E('**⟨*rule*⟩**')'**

⟨*disabled*⟩→ **'D('**⟨*rule*⟩**')'**

⟨*custom*⟩→ **'['** [⟨*navigation*⟩ | ⟨*logical*⟩] **']'**

⟨*constant*⟩→ **'C('**⟨*number*⟩**')'**

⟨*number*⟩→ [0-9]+

⟨*logNumOp*⟩→ '=' | '≠' | '<' | '>' | '≤' | '≥'

⟨*logBinOp*⟩→ '∧' | '∨' | '⊕'

⟨*logUnOp*⟩→ '¬'

⟨*navOp*⟩→ ['◄' | '►'] ['+' | ⟨*constant*⟩]?

⟨*numOp*⟩→ **'+'** | **'-'** | **'*'** | **'/'**

B Evaluation algorithm

The following is the pseudocode for the criteria evaluation algorithm:

function EVALUATE(Cr,G_d) ▷ Evaluation of criteria
  let $L_n \leftarrow \phi$ ▷ Ordered list of nodes
  let $S_{st} \leftarrow s : startingPoints \in Cr$ ▷ Gather starting points
  let $S \leftarrow S_{st}[1]$ ▷ Select first starting point
  for $i \leftarrow 2, size(S_{st})$ **do** ▷ Check all other starting points
    if $S \neq S_{st}[i]$ **then** ▷ Two starting point is equal, if they are of the same type
      let $inconsistent \leftarrow true$ ▷ Inconsistent criteria cannot be evaluated
    end if
  end for
  if $\neg inconsistent$ **then**
    let $S_n \leftarrow n \in G_d | S(n) \leftarrow true$ ▷ Gather nodes satisfying starting point
    let $O \leftarrow enclosing(S)$ ▷ Get enclosing operator
    for all $n \in S_n$ **do** ▷ Iterate through gathered nodes
      let $N_c \leftarrow n, N_v \leftarrow \phi$ ▷ Initialize set of current and visited nodes
      let $R_p \leftarrow 0, R_b \leftarrow false, limit \leftarrow -1, break \leftarrow false$ ▷ Initialize partial results, navigation limit and loop break signal
      while $O \neq Cr$ **do** ▷ Exit loop as the criteria on the given node is evaluated
        if $O = limited \wedge limit = -1$ **then**

```
            limit ← limit(O)                                          ▷ Set limit for limited operators
        end if
        APPLYOPERATOR(O,G_d,N_c,N_v,R_p,R_b,L_n)                      ▷ Apply operator
        if R_p = true ∧ Cr = cut-off ∧ quantifier(Cr) = existential then   ▷ Check cut-off condition for existential
            return R_b                                                ▷ Terminate evaluation with cut-off
        else
            if Cr = cut-off ∧ quantifier(Cr) = universal then         ▷ Check cut-off condition for universal
                return R_b                                            ▷ Terminate evaluation without cut-off
            else
                break ← true
                O ← Cr                                                ▷ Force exit from loop at next check
            end if
        end if
        if O = limited ∧ N_c ≠ φ ∧ limit > 1 then                     ▷ If further iterations are required
            limit ← limit − 1                                         ▷ Decrease limit
        else if O ≠ transitive ∨ N_c ≠ φ then                         ▷ Not transitive navigation or no new reached nodes
            O ← enclosing(O)                                          ▷ Get enclosing operator
            limit ← −1                                                ▷ Reset limit
        end if                                                        ▷ Otherwise continue transitive navigation, if new nodes were reached
    end while
    if Cr = selection ∧ break = false then                            ▷ If the criterion is completely evaluated
        L_n ← L_n ∩ (R_p, n)                                          ▷ Add result and node to list
        R_b ← false
    end if
  end for
  if R_b = false then
    if Cr = selection then
      if ordering(Cr) = min then
        L_n ← reverse(L_n)                                            ▷ Reverse order for min
      end if
      return L_n                                                      ▷ Return result list for selection
    else
      return R_p                                                      ▷ Return without cut-off
    end if
  else
    return R_b                                                        ▷ Return with cut-off
  end if
 else
  ERROR                                                               ▷ Starting points inconsistent
 end if
end function
function APPLYOPERATOR(O,G_d,N_c,N_v,R_p,R_b,L_n)                      ▷ Application of an operator
 if O ∈ navOp then                                                    ▷ Navigation functions
  let N_r ← reach(O, N_c, G_d) \ N_v   ▷ Gather reachable states from dependency graph, excluding already visited nodes
  for all n ∈ N_r do
    R_p ← R_p + #_n                                                   ▷ Update partial result with remaining executions
  end for
  N_v ← N_v ∩ N_r                                                     ▷ Update set of visited nodes
  N_c ← N_r                                                           ▷ Update set of current nodes
 else if O ∈ logUnOp then                                             ▷ Unary logical functions
  R_b ← op(O) R_b
 else
  let right ← EVALUATE(right(O), G_d)                                 ▷ Evaluate right side of criteria
  if O ∈ logNumOp then                                                ▷ Logical functions on numbers
    R_b ← R_p op(O) right                                             ▷ Update boolean result with evaluated value
  else if O ∈ logBinOp then                                           ▷ Binary logical functions
    R_b ← R_b op(O) right
  else if O ∈ numOp then                                              ▷ Numerical functions
    R_p ← R_p op(O) right                                             ▷ Update partial result with evaluated value
  end if
 end if
end function
```